Context-free Grammars

- Set of rules by which valid sentences can be constructed.
- Example:
  - Sentence $\rightarrow$ Noun Verb Object
  - Noun $\rightarrow$ trees $|$ parsers
  - Verb $\rightarrow$ are $|$ grow
  - Object $\rightarrow$ on Noun $|$ Adjective
  - Adjective $\rightarrow$ slowly $|$ interesting
- What strings can Sentence derive?
- Syntax only – no semantic checking
Derivations of a CFG

- parsers grow on trees
- parsers grow on Noun
- parsers grow Object
- parsers Verb Object
- Noun Verb Object
- Sentence

Derivations and parse trees
Arithmetic Expressions

- $E \rightarrow E + E$
- $E \rightarrow E \times E$
- $E \rightarrow (E)$
- $E \rightarrow -E$
- $E \rightarrow \text{id}$

Leftmost derivations for $\text{id} + \text{id} \times \text{id}$

- $E \Rightarrow E + E$
- $E \Rightarrow \text{id} + E$
- $E \Rightarrow \text{id} + E \times E$
- $E \Rightarrow \text{id} + \text{id} \times E$
- $E \Rightarrow \text{id} + \text{id} \times \text{id}$
Leftmost derivations for \( \text{id} + \text{id} \ast \text{id} \)

\[
\begin{align*}
E & \rightarrow E + E \\
E & \rightarrow E \ast E \\
E & \rightarrow (E) \\
E & \rightarrow -E \\
E & \rightarrow \text{id}
\end{align*}
\]

• \( E \Rightarrow E \ast E \)

\( \Rightarrow E + E \ast E \)

\( \Rightarrow \text{id} + E \ast E \)

\( \Rightarrow \text{id} + \text{id} \ast E \)

\( \Rightarrow \text{id} + \text{id} \ast \text{id} \)


Rightmost derivation for \( \text{id} + \text{id} \ast \text{id} \)

\[
\begin{align*}
E & \rightarrow E + E \\
E & \rightarrow E \ast E \\
E & \rightarrow (E) \\
E & \rightarrow -E \\
E & \rightarrow \text{id}
\end{align*}
\]

\( E \Rightarrow E + E \)

\( \Rightarrow E + E \ast E \)

\( \Rightarrow \text{id} + E \ast E \)

\( \Rightarrow \text{id} + \text{id} \ast \text{id} \)

\( \Rightarrow \text{id} + \text{id} \ast \text{id} \)
Rightmost derivation for \(\text{id} + \text{id} \ast \text{id}\)

\[
\begin{align*}
E & \rightarrow E + E \\
E & \rightarrow E \ast E \\
E & \rightarrow (E) \\
E & \rightarrow -E \\
E & \rightarrow \text{id}
\end{align*}
\]

Parsing - Roadmap

- Parser is a decision procedure: builds a parse tree
- Top-down vs. bottom-up
- Recursive-descent with backtracking
- Bottom-up parsing (CKY)
- Shift-reduce parsing
- Combining top-down and bottom-up: Earley parsing
Top-Down vs. Bottom Up

Grammar:  \[ S \rightarrow A \ B \]
\[ A \rightarrow c \mid \epsilon \]
\[ B \rightarrow cbB \mid ca \]

Input String: ccba

Input String: ccba

Top-Down/leftmost | Bottom-Up/rightmost
---|---
S ⇒ AB | S⇒ AB
⇒ cB | A⇒ c
⇒ ccB | B⇒ cbB
⇒ ccba | B⇒ ca

S⇒ AB
⇒ A⇒ c
⇒ B⇒ cbB
⇒ B⇒ ca

ccba ← Acbca
⇐ AcB
⇐ AB
⇐ S

A⇒ c
B⇒ ca
B⇒ cbB
S⇒ AB

Top-Down: Backtracking

True/False

S ⇒* ccba?

S     cbca try S⇒ AB
AB    cbca try A⇒ c
cB    cbca match c
B     bca dead-end, try A⇒ ε
cB    cbca try B⇒ cbB
cbB   cbca match c
bB    bca match b
B     ca try B⇒ cbB
cbB   ca match c
bB    a dead-end, try B⇒ ca
cbB   a match c
ca    a match a, Done!
Transition Diagram

S $\rightarrow$ cAa

A $\rightarrow$ cB $|$ B

B $\rightarrow$ bcB $|$ $\epsilon$

Bottom-up parsing overview

- Start from terminal symbols, search for a path to the start symbol
- Apply shift and reduce actions: postpone decisions
- LR parsing:
  - L: left to right parsing
  - R: rightmost derivation (in reverse or bottom-up)
- Useful for deterministic parsing (e.g. in compilers for programming languages)
Rightmost derivation for \( \text{id} + \text{id} * \text{id} \)

\[
\begin{align*}
E & \rightarrow E + E & E & \rightarrow E * E \\
E & \rightarrow E * E & \Rightarrow & E \ast \text{id} \\
E & \rightarrow (E) & \Rightarrow & E + E \ast \text{id} \\
E & \rightarrow -E & \Rightarrow & E + \text{id} \ast \text{id} & \text{(reduce with } E \rightarrow \text{id}) \\
E & \rightarrow \text{id} & \Rightarrow & \text{id} + \text{id} \ast \text{id} & \text{(shift)}
\end{align*}
\]

Ambiguity

- Grammar is ambiguous if more than one parse tree is possible for some sentences
- Examples in English:
  - Two sisters reunited after 18 years in checkout counter
- It is undecidable to check using an algorithm whether a grammar is ambiguous
Parsing CFGs

- Consider the problem of parsing with arbitrary CFGs
- For any input string, the parser has to produce a parse tree
- The simpler problem: print \textbf{yes} if the input string is generated by the grammar, print \textbf{no} otherwise
- This problem is called \textit{recognition}

CKY Recognition Algorithm

- The Cocke-Kasami-Younger algorithm
- As we shall see it runs in time that is polynomial in the size of the input
- It takes space polynomial in the size of the input
- \textbf{Remarkable fact:} it can find all possible parse trees (exponentially many) in polynomial time
Chomsky Normal Form

• Before we can see how CKY works, we need to convert the input CFG into Chomsky Normal Form
• CNF is one of many grammar transformations that preserve the language
• CNF means that the input CFG G is converted to a new CFG G’ in which all rules are of the form:
  A → B C
  A → a

Epsilon Removal

• First step, remove epsilon rules
  A → B C
  C → ε | C D | a
  D → b  B → b
• After ε-removal:
  A → B | B C D | B a | BC
  C → D | C D D | a D | C D | a
  D → b  B → b
Removal of Chain Rules

• Second step, remove chain rules
  \[ A \rightarrow B \quad C \mid C \quad D \quad C \]
  \[ C \rightarrow D \mid a \]
  \[ D \rightarrow d \quad B \rightarrow b \]

• After removal of chain rules:
  \[ A \rightarrow B \quad a \mid B \quad D \mid a \quad D \mid a \quad D \mid D \mid D \quad d \mid D \quad a \mid D \quad D \quad D \]
  \[ D \rightarrow d \quad B \rightarrow b \]

Eliminate terminals from RHS

• Third step, remove terminals from the rhs of rules
  \[ A \rightarrow B \quad a \quad C \quad d \]

• After removal of terminals from the rhs:
  \[ A \rightarrow B \quad N_1 \quad C \quad N_2 \]
  \[ N_1 \rightarrow a \]
  \[ N_2 \rightarrow d \]
Binarize RHS with Nonterminals

- Fourth step, convert the rhs of each rule to have two non-terminals
  \[ A \rightarrow B N_1 C N_2 \]
  \[ N_1 \rightarrow a \]
  \[ N_2 \rightarrow d \]
- After converting to binary form:
  \[ A \rightarrow B N_3 \]
  \[ N_1 \rightarrow a \]
  \[ N_3 \rightarrow N_1 N_4 \]
  \[ N_2 \rightarrow d \]
  \[ N_4 \rightarrow C N_2 \]

CKY algorithm

- We will consider the working of the algorithm on an example CFG and input string
- Example CFG:
  \[ S \rightarrow A X \mid Y B \]
  \[ X \rightarrow A B \mid B A \]
  \[ Y \rightarrow B A \]
  \[ A \rightarrow a \]
  \[ B \rightarrow a \]
- Example input string: aaa
### CKY Algorithm

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A, B</td>
<td>X, Y</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A → a</td>
<td>X → A B</td>
<td>S → A(0,1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B → a</td>
<td>B → a</td>
<td>Y → B A</td>
<td>S → Y(0,2)</td>
</tr>
<tr>
<td>1</td>
<td>A, B</td>
<td>X, Y</td>
<td>A, B</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A → a</td>
<td>X → A B</td>
<td>X → A B A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B → a</td>
<td>B → a</td>
<td>Y → B A</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>A, B</td>
<td>X, Y</td>
<td>A, B</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A → a</td>
<td>X → A B</td>
<td>A → a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B → a</td>
<td>B → a</td>
<td>B → a</td>
<td></td>
</tr>
</tbody>
</table>

### Parse trees

- Tree 1: 
  - S
  - Y
  - B
  - a
  - a
  - a

- Tree 2: 
  - S
  - A
  - X
  - A
  - B
  - a
  - a
  - a

- Tree 3: 
  - S
  - A
  - X
  - B
  - A
  - a
  - a
  - a
CKY Algorithm

Input string input of size n
Create a 2D table chart of size $n^2$
for i=0 to n-1
    chart[i][i+1] = A if there is a rule $A \rightarrow a$ and input[i]=a
for j=2 to N
    for i=j-2 downto 0
        for k=i+1 to j-1
            chart[i][j] = A if there is a rule $A \rightarrow B C$ and chart[i][k] = B and chart[k][j] = C
    return yes if chart[0][n] has the start symbol
else return no

CKY algorithm summary

• Parsing arbitrary CFGs
• For the CKY algorithm, the time complexity is $O(|G|^2 n^3)$
• The space requirement is $O(n^2)$
• The CKY algorithm handles arbitrary ambiguous CFGs
• All ambiguous choices are stored in the chart
• For compilers we consider parsing algorithms for CFGs that do not handle ambiguous grammars
Parsing - Summary

• Parsing arbitrary CFGs: $O(n^3)$ time complexity
• Top-down vs. bottom-up
  – Recursive-descent parsing
  – Shift-reduce parsing
• Earley parsing
• Ambiguous grammars result in parser output with multiple parse trees for a single input string

Parsing - Additional Results

• $O(n^2)$ time complexity for linear grammars
  – All rules are of the form $S \rightarrow aSb$ or $S \rightarrow a$
  – Reason for $O(n^2)$ bound is the linear grammar normal form: $A \rightarrow aB$, $A \rightarrow Ba$, $A \rightarrow B$, $A \rightarrow a$
• Left corner parsers
  – extension of top-down parsing to arbitrary CFGs
• Earley’s parsing algorithm
  – $O(n^3)$ worst case time for arbitrary CFGs just like CKY
  – $O(n^2)$ worst case time for unambiguous CFGs
  – $O(n)$ for specific unambiguous grammars
    (e.g. $S \rightarrow aSa \mid bSb \mid \varepsilon$)
Non-CF Languages

\[ L_1 = \{w cw \mid w \in (a|b)^*\} \]
\[ L_2 = \{a^n b^m c^n d^m \mid n \geq 1, m \geq 1\} \]
\[ L_3 = \{a^n b^n c^n \mid n \geq 0\} \]

CF Languages

\[ L_4 = \{w cw^R \mid w \in (a|b)^*\} \]
\[ S \to aSa \mid bSb \mid c \]
\[ L_5 = \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\} \]
\[ S \to aSd \mid aAd \]
\[ A \to bAc \mid bc \]
Context-free languages and Pushdown Automata

- Recall that for each regular language there was an equivalent finite-state automaton
- The FSA was used as a recognizer of the regular language
- For each context-free language there is also an automaton that recognizes it: called a pushdown automaton (pda)

Pushdown Automata

- PDA has
  - an alphabet (terminals) and
  - stack symbols (like non-terminals),
  - a finite-state automaton, and
  - stack

\[
\begin{align*}
\epsilon, \epsilon & \rightarrow S \\
0, \epsilon & \rightarrow A \\
1, A & \rightarrow \epsilon \\
\epsilon, S & \rightarrow \epsilon \\
1, A & \rightarrow \epsilon
\end{align*}
\]

- e.g. PDA for language \( L = \{ 0^n1^n : n \geq 0 \} \)
  - \( \rightarrow \) implies a push/pop of stack symbol(s)
  - push stack symbol A
  - pop stack symbol A
  - check that stack is empty
Shift-reduce parser as a pda

Non-deterministic PDA that is a parser for grammar: $S := 0S1 | 2$
$L(S) = \{ 0^n 2 1^n : n \geq 0 \}$

- $\varepsilon, \varepsilon \rightarrow \$\quad 0, \varepsilon \rightarrow 0$
- $2, \varepsilon \rightarrow S\quad \varepsilon, 0S1 \rightarrow S$
- $\varepsilon, S$ $\rightarrow \varepsilon\quad 1, \varepsilon \rightarrow 1$

check that stack is empty

Reduce action

$\quad$→ implies a push/pop of stack symbol(s)

2/29/08

Context-free languages and Pushdown Automata

- Similar to FSAs there are non-deterministic pda and deterministic pda
- Unlike in the case of FSAs we cannot always convert a npda to a dpda
- The construction of a pda will provide us with the algorithm for parsing (take in strings and provide the parse tree)
CKY algorithm for PCFGs

- We will consider the working of the algorithm on an example PCFG and input string.
- Example PCFG:
  \[ S \rightarrow A \, X \, (0.3) \mid Y \, B \, (0.7) \]
  \[ X \rightarrow A \, B \, (0.1) \mid B \, A \, (0.9) \]
  \[ Y \rightarrow B \, A \, (1.0) \]
  \[ A \rightarrow a \, (1.0) \]
  \[ B \rightarrow a \, (1.0) \]
- Example input string: \textit{aaa}
Parse trees

PCFG is consistent:
\[ 0.7 + 0.27 + 0.03 = 1.0 \]